Розробка мобільних додатків

Лекція 2 - Знайомство з основними компонентами

React vs. React Native

React — це бібліотека для створення веб-інтерфейсів. **React Native** — це фреймворк для розробки мобільних додатків.

- **Цільова платформа:**
  - **React:** Браузери (Chrome, Safari, Firefox). Працює з моделлю **DOM**.
  - **React Native:** Мобільні ОС (iOS, Android). Працює через **Native API**.
- **Механізм відображення:**
  - **React:** Використовує *Virtual DOM* для оновлення HTML-дерева.
  - **React Native:** Використовує *Bridge* (або новий JSI) для виклику нативних компонентів системи.
- **Результат:**
  - У вебі ви отримуєте набір вузлів у браузері.
  - У мобільному додатку ви отримуєте реальні системні віджети (кнопки, списки, поля вводу), які не відрізняються від написаних на Swift чи Kotlin.

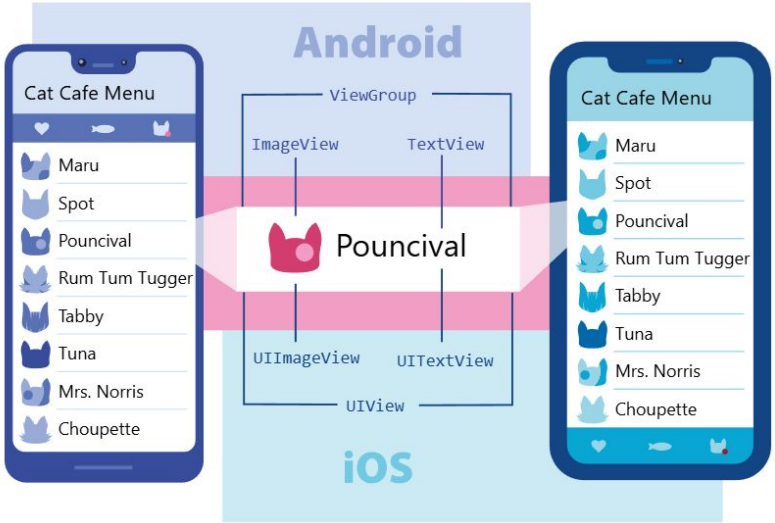
React vs. React Native

В React Native ми відмовляємося від візуальної мови вебу (HTML) на користь **абстракцій над нативними компонентами ОС**. Кожен базовий компонент React Native має свій відповідник у коді Swift (iOS) та Kotlin/Java (Android).

Механізм "Мапінгу"

React Native дає команду системі створити нативний елемент. Наприклад, компонент `<View>` під час рендерингу перетворюється на:

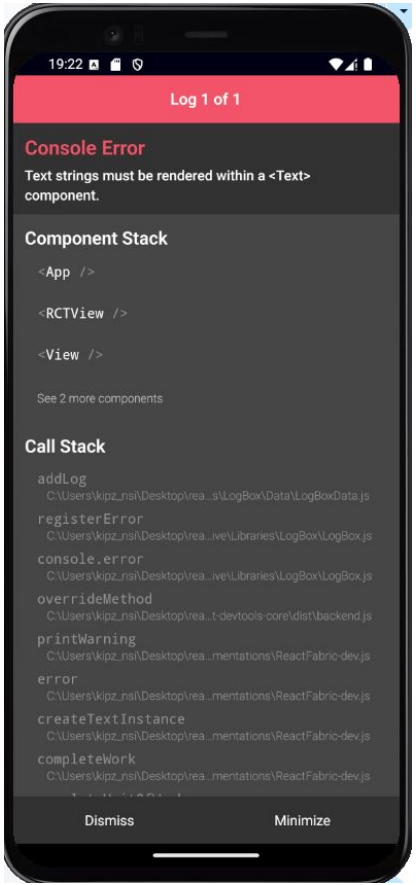
- **iOS:** `UIView`
- **Android:** `android.view.ViewGroup`



<Text>

У веб-розробці будь-який HTML-тег може містити текстовий вузол. У React Native правила значно суворіші через особливості рендерингу.

Будь-який текст має бути явно вкладений у компонент `<Text>`



<Text>

Оскільки в React Native немає тегів `<span>`, `<b>` або `<i>`, компонент `<Text>` бере на себе всі ролі з оформлення текстового контенту, використовуючи механізм вкладеності.

Вкладеність та успадкування

Це єдиний випадок у React Native, де працює **успадкування стилів**. Якщо один `<Text>` знаходиться всередині іншого, дочірній елемент успадкує стилі батьківського.

- **Приклад:** ```jsx <Text style={{ color: 'blue' }}> приклад, <Text style={{ fontWeight: 'bold' }}>приклад</Text> </Text>
- **Особливість:** Вкладені тексти не створюють нових рядків (блок-контейнерів), вони рендеряться в один рядок, як інлайнові елементи у вебі.

<Text>

Ключові властивості (Props)

Text · React Native

Компонент має специфічні для мобільних пристроїв атрибути, яких немає в HTML:

1. **`numberOfLines`**: Обмежує текст певною кількістю рядків. Якщо тексту більше — він обрізається.
2. **`ellipsizeMode`**: Визначає, де ставити трикрапку при обрізанні (`head`, `middle`, `tail`). За замовчуванням — `tail` (в кінці).
3. **`selectable`**: Визначає, чи може користувач виділити та скопіювати текст (за замовчуванням `false`, на відміну від браузера).
4. **`onPress`**: Дозволяє зробити частину тексту (або весь блок) клікабельним (наприклад, посилання "Читати далі").

Text Style Props

https://reactnative.dev/docs/text-style-props?language=javascript

React vs. React Native

У React Native відсутня концепція каскадних таблиць стилів (CSS). Замість цього використовується механізм **StyleSheet**, який транслює JavaScript-об'єкти у властивості нативних графічних рушіїв iOS та Android.

StyleSheet API та його переваги

Замість маніпуляцій з класами, ми використовуємо `StyleSheet.create()`.

- **Продуктивність:** Стилі відправляються до нативної частини лише один раз, а не при кожному рендерингу.
- **Валідація:** Помилки в назвах властивостей відловлюються на етапі компіляції, а не в рантаймі.
- **Ізоляція:** Стилі не є глобальними. Це вирішує проблему "конфлікту імен", яка притаманна великим веб-проєктам.

StyleSheet vs CSS — Ключові відмінності

StyleSheet · React Native

1. Відсутність каскадності

- **Web:** Стилі успадковуються від батьківських елементів до дочірніх (наприклад, `font-family` для `body` застосується до всіх тегів).
- **Native:** Кожен компонент ізольований. Колір, заданий у `<View>`, не вплине на `<Text>` всередині нього. Кожен елемент має бути стилізований явно.

2. CamelCase замість Kebab-case

У React Native ми працюємо з властивостями об'єктів JS, тому назви пишуться без дефісів:

- **CSS:** `background-color`, `font-size`, `margin-top`.
- **RN:** `backgroundColor`, `fontSize`, `marginTop`.

StyleSheet vs CSS — Ключові відмінності

3. Числові значення замість одиниць вимірювання

- **Web:** Обов'язково вказувати одиниці: `16px`, `2rem`, `10%`, `50vh`.
- **Native:** Використовуються **числа без одиниць**. Це "логічні пікселі", які система автоматично масштабує під щільність екрана (DPI) пристрою.*Виняток:* Рядкові значення підтримуються лише для відсотків (наприклад, `width: '50%'`).

4. Обмежена підмножина властивостей

React Native не підтримує 100% властивостей CSS. Наприклад, ви не знайдете:

- `grid` (замість нього тільки Flexbox).
- Псевдоселектори (`:hover`, `:nth-child`, `:before`).
- Складні фільтри та деякі типи анімацій, що притаманні лише рушіям браузерів (WebKit/Blink).

5. Помилки та валідація

- **Web:** Якщо ви напишете `color: "червоний"`, браузер просто проігнорує цей рядок.
- **Native:** `StyleSheet.create` проводить валідацію. Невірна назва властивості або неправильний тип значення (наприклад, рядок замість числа) призведе до попередження або помилки ще на етапі розробки.

DPI

Це фундаментальне поняття в мобільній розробці. Якщо у вебі ми звикли до пікселів, то в мобільних додатках ми працюємо з **абстрактними одиницями**, щоб інтерфейс виглядав однаково і на старому смартфоні, і на сучасному флагмані.

Що таке DPI та PPI?

- **DPI (Dots Per Inch)** — кількість точок на один дюйм. Термін прийшов із поліграфії, але в мобайлі він означає щільність пікселів.
- **PPI (Pixels Per Inch)** — фактична кількість фізичних пікселів, які виробник вмістив у один дюйм екрана.

Проблема: Якщо ви зробите кнопку розміром 100x100 фізичних пікселів:

1. На екрані з низькою щільністю вона буде величезною.
2. На екрані з високою щільністю (Retina/Super AMOLED) вона буде крихітною, бо пікселі там набагато менші за розміром.

Властивість **style vs className**

В React Native ми повністю відмовляємося від атрибута `className`. Замість того, щоб посилатися на зовнішню таблицю стилів через рядковий ідентифікатор (клас), ми передаємо безпосередньо **об'єкт JavaScript**.

1. Чому не `className`?

- **Відсутність DOM:** У мобільних додатках немає HTML-дерева та CSS-парсера, який міг би шукати правила за назвою класу в глобальному просторі.

2. Механізм роботи `style={styles.title}`

Коли ми пишемо `style={styles.title}`, ми передаємо компоненту посилання на конкретний об'єкт із набору, створеного через `StyleSheet.create`.

- **`styles`** — це об'єкт-контейнер.
- **`title`** — це ключ, який містить набір нативних властивостей.

Кольори у **React Native**

React Native підтримує більшість стандартних форматів кольорів, які використовуються у вебі.

Підтримувані формати:

- **Назви кольорів:** `red`, `blue`, `transparent` (стандартні іменовані кольори HTML).
- **HEX-коди:** `'#FFFFFF'`, `'#f0f'`, `'#ff00ff00'` (останні дві цифри — це прозорість/alpha).
- **RGB / RGBA:** `'rgb(255, 0, 0)'` або `'rgba(255, 0, 0, 0.5)'`.
- **HSL / HSLA:** `'hsl(360, 100%, 50%)'`.

Color Reference · React Native

<View> у React Native

У React Native компонент **<View>** — це будівельний блок для створення інтерфейсу. Він є прямим аналогом тегу `<div>` у веб-розробці, але з нативною реалізацією.

Використання:

- **Групування елементів:** Використовується як контейнер для об'єднання тексту, зображень, кнопок та інших компонентів у логічні блоки.
- **Стилізація та оформлення:** Дозволяє задавати візуальні властивості: колір фону (`backgroundColor`), внутрішні відступи (`padding`), межі та закруглення (`borderRadius`).
- **Макетування (Flexbox):** Слугує основою для розташування елементів на екрані. Кожен `<View>` за замовчуванням є Flex-контейнером.
- **Створення компонентів:** Слугує для побудови складніших елементів інтерфейсу, таких як картки, списки, навігаційні панелі або модальні вікна.

Важливі нюанси:

1. **Без скролінгу:** Якщо контент не вміщується в межі `<View>`, він просто обрізається. Для прокрутки використовується спеціальний компонент `ScrollView`.

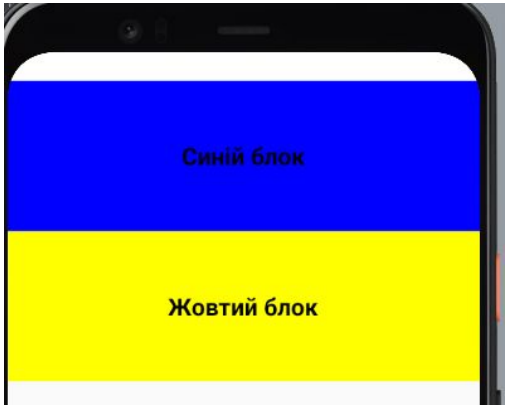
Особливості <View>

<View> – це блоковий контейнер

У React Native всі <View> працюють як **блокові елементи**, тобто займають весь доступний простір по ширині (аналог display: block у CSS).

Приклад: два <View> будуть розташовані вертикально:

```
<>
    <View style={[styles.block, { backgroundColor: 'blue' }]}>
        <Text style={styles.text}>Синій блок</Text>
    </View>
    <View style={[styles.block, { backgroundColor: 'yellow' }]}>
        <Text style={styles.text}>Жовтий блок</Text>
    </View>
</>
```

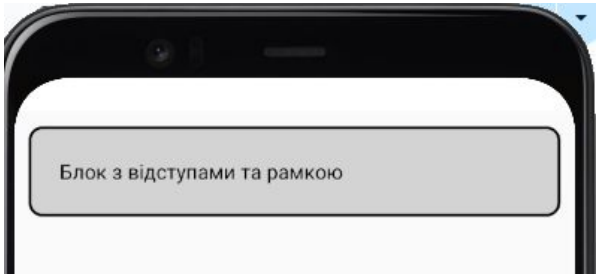


Особливості <View>

Використання відступів та меж (`margin`, `padding`, `border`)

Як і в CSS, у `<View>` можна задавати **зовнішні (margin) та внутрішні (padding) відступи**, а також границі (`border`):

```
<View style={{
  backgroundColor: 'lightgray',
  padding: 20,
  margin: 10,
  borderWidth: 2,
  borderColor: 'black',
  borderRadius: 10
}}>
  <Text>Блок з відступами та рамкою</Text>
</View>
```



<View>

View Style Props · React Native

Layout with Flexbox у React Native

Flexbox (Flexible Box Layout) — це універсальна система побудови інтерфейсів у React Native, що дозволяє декларативно керувати розподілом простору, розмірами та вирівнюванням елементів усередині контейнерів.

Якщо у вебі це одна з систем верстки, то тут це **єдина система розкладки інтерфейсу**..

В основі роботи лежить рушій **Yoga** — кросплатформний алгоритм обчислення layout, розроблений для інтерфейсів із декларативним описом структури. Yoga не відповідає за відмальовування елементів, а виконує геометричні розрахунки: визначає розміри компонентів і їхнє розташування на екрані відповідно до заданих правил Flexbox.

Flexbox у React Native концептуально подібний до CSS Flexbox, проте має низку особливостей, зумовлених мобільним середовищем:

Основи **Flexbox** у **React Native**

Усі елементи у `<View>` автоматично мають `display: flex` (не потрібно вказувати вручну).

Використовуються три основні параметри:

- `flexDirection` (напрямок осі)
- `justifyContent` (вирівнювання по головній осі)
- `alignItems` (вирівнювання по поперечній осі)

flexDirection – напрямок осі

`flexDirection` визначає напрямок розташування дочірніх елементів.

Відмінність від вебу: У React Native **за замовчуванням** `flexDirection: 'column'`, а не `'row'`

```jsx
export default function App() {
  return (
      <View style={styles.container}>
        <Text>1</Text>
        <Text>2</Text>
        <Text>3</Text>
      </View>
  );
}

const styles : {container: {...}} = StyleSheet.create({
  container: {
    flexDirection: 'row', // Горизонтальне розташування
    backgroundColor: 'lightgray',
  },
});
```

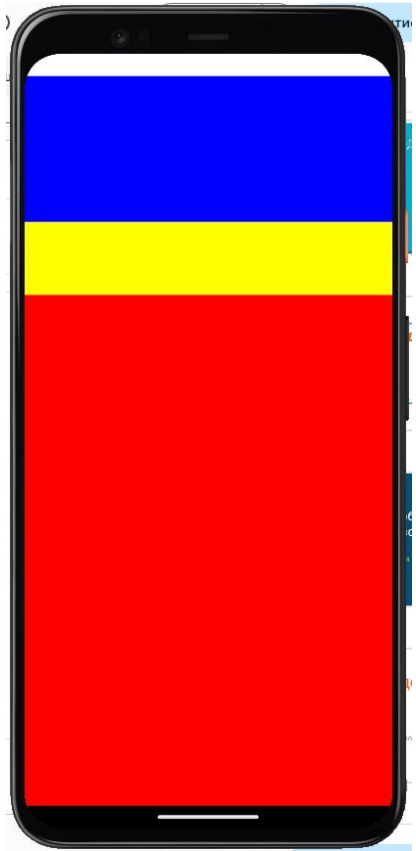
Властивість `flex`

Властивість `flex` визначає, як компонент має розширюватися, щоб заповнити доступне місце в батьківському контейнері.

- **`flex: 1`**: Каже компоненту зайняти **весь** вільний простір. Якщо таких компонентів декілька, вони розділять простір порівну.
- **Пропорції:** Якщо одному блоку дати `flex: 2`, а іншому `flex: 1`, то перший забере 2/3 екрана, а другий — 1/3.

Значення `flex`	Опис
`0` (за замовчуванням)	Елемент має фіксовану ширину/висоту
`1`	Заповнює весь доступний простір
`2, 3, ...`	Елементи ділять простір пропорційно

```jsx
export default function App() {
  return (
      <View style={styles.container}>
        <View style={styles.box1}></View>
        <View style={styles.box2}></View>
      </View>
  );
}

const styles {...} = StyleSheet.create({
  container: {
    flex: 1, // Займає весь екран
    backgroundColor: 'red',
  },
  box1: {
    flex: 0.2, // Займає більше простору
    backgroundColor: 'blue',
  },
  box2: {
    flex: 0.1, // Займає менше простору
    backgroundColor: 'yellow',
  },
});
```



[Layout with Flexbox · React Native]

<TextInput>

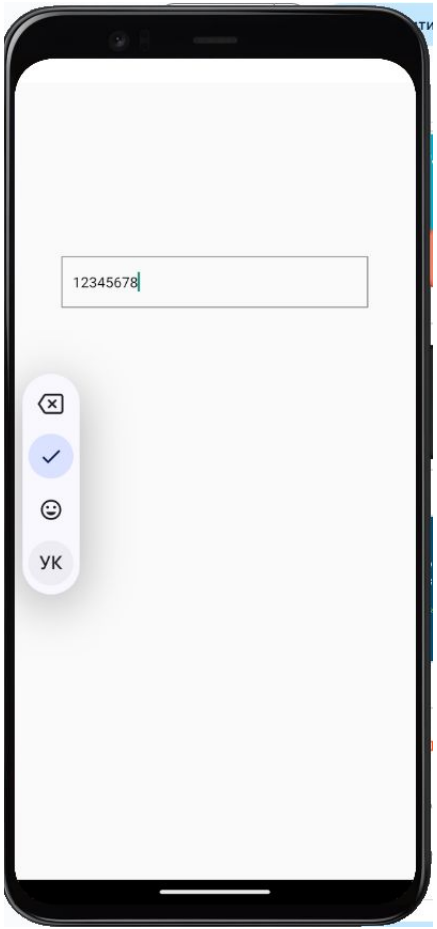
`<TextInput>` — це базовий компонент для введення тексту за допомогою екранної клавіатури. Він дозволяє обробляти як короткі рядки (логін, пароль), так і багаторядкові тексти (коментарі).

Основні властивості:

- **`value`**: Поточне значення в полі (контрольований компонент).
- **`onChangeText`**: Функція, яка спрацьовує при зміні тексту. Вона приймає сам текст як аргумент (а не об'єкт події `e.target.value`).
- **`placeholder`**: Текст-підказка, який зникає при введенні.
- **`secureTextEntry`**: Якщо `true`, текст замінюється на крапки (використовується для паролів).

TextInput · React Native

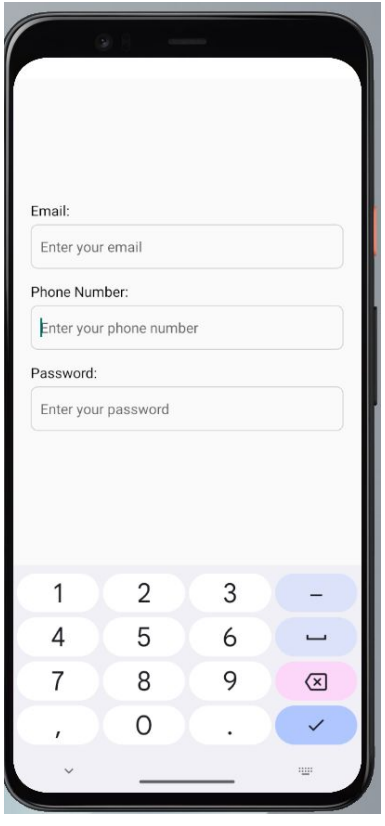
```tsx
export default function App() {
  const [text : string , setText] = useState( initialState: '');

  return (
      <View style={styles.container}>
        <TextInput
            style={styles.input}
            placeholder="Введіть текст..."
            value={text}
            onChangeText={setText}
        />
      </View>
  );
}

const styles : {container: {...}, input: {...}} = StyleSheet.create({
  container: {
    flex: 0.5,
    alignItems: "center",
    justifyContent: 'center',
  },
  input: {
    width: 300,
    height: 50,
    borderColor: 'gray',
    borderWidth: 1,
    paddingHorizontal: 10
  },
});
```



Керування клавіатурою

Особливість мобільної розробки. Ви можете підказати системі, яку клавіатуру відкрити:

- **`keyboardType`**: Може бути `default`, `numeric` , `email-address` або `phone-pad`.
- **`returnKeyType`**: Змінює напис на клавіші "Enter" (наприклад: `done`, `go`, `next`, `search`).



```jsx
{/* Numeric Input */}
<Text style={styles.label}>Phone Number:</Text>
<TextInput
    style={styles.input}
    keyboardType="numeric"
    placeholder="Enter your phone number"
    value={phone}
    onChangeText={setPhone}
/>

{/* Password Input */}
<Text style={styles.label}>Password:</Text>
<TextInput
    style={styles.input}
    secureTextEntry={true}
    placeholder="Enter your password"
    value={password}
    onChangeText={setPassword}
/>
```

Багаторядковий ввід

У React Native відсутній поділ на `<input>` та `<textarea>`. Компонент `<TextInput>` — це єдиний інструмент, який стає багаторядковим при додаванні пропса `multiline={true}`

Зміна поведінки через `multiline`

Щоб перетворити звичайне поле вводу на аналог `textarea`, достатньо додати булеву властивість:

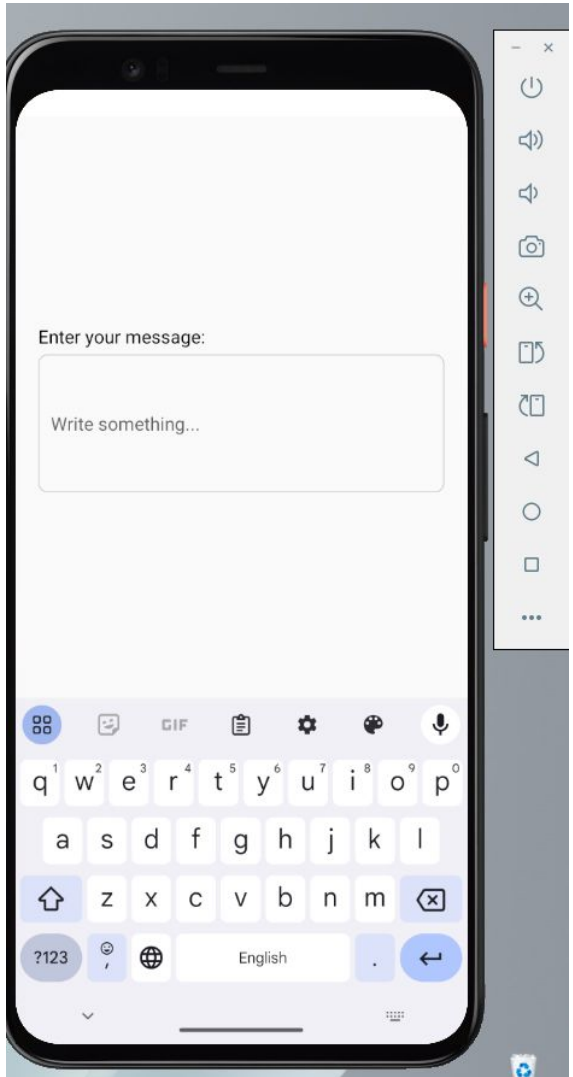
- **`multiline={true}`**: Дозволяє тексту переноситися на нові рядки та розширює функціонал клавіші "Enter" (вона починає створювати новий рядок, а не завершувати ввід).

Керування розміром

Замість атрибута `rows`, у React Native ми використовуємо:

- **`numberOfLines={4}`**: (для Android) Встановлює початкову видиму висоту компонента.
- **Стилізація `height`**: Для повного контролю висоти на обох платформах краще використовувати `height` або `minHeight` у об'єкті `StyleSheet`.

```jsx
<View style={styles.container}>
    <Text style={styles.label}>Enter your message:</Text>
    <TextInput
        style={styles.textarea}
        placeholder="Write something..."
        value={text}
        onChangeText={setText}
        multiline={true} // Дозволяє вводити кілька рядків
        numberOfLines={4} // Висота за замовчуванням (кількість рядків)
    />
</View>
```



На Android текст у багаторядковому полі за замовчуванням може центруватися по вертикалі. Щоб він починався з верхнього лівого кута (як у класичному `textarea`), обов'язково додавайте стиль: textAlignVertical: 'top'

Кнопки у **React Native**

У React Native **немає стандартного HTML-елемента `<button>`**, тому для створення кнопок використовуються **нативні компоненти**:

1. `<Button>` – **простий, стандартний компонент кнопки**.
2. `<TouchableOpacity>` – кнопка, яка міняє прозорість при натисканні.
3. `<TouchableHighlight>` – кнопка, яка затемнюється при натисканні.
4. `<Pressable>` – новий API для контролю взаємодії.

Головна відмінність від вебу:

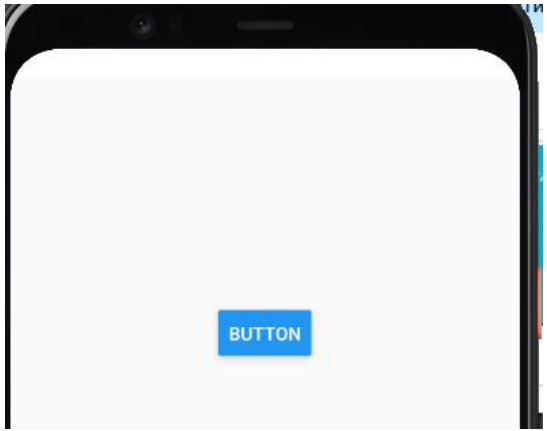
- Всі кнопки – це **нативні компоненти**, що працюють на iOS та Android по-різному.
- `Button` обмежений у стилізації

<Button> – Простий варіант кнопки

`<Button>` — це найпростіший інтерактивний компонент, який використовує стандартні стилі операційної системи (iOS або Android). Це робить кнопку впізнаваною для користувача, але створює значні обмеження для дизайнера.

Критичні обмеження `<Button>`:

- **Відсутність кастомізації:** Ви не можете змінити внутрішні відступи (`padding`), закруглення кутів (`borderRadius`), шрифт або тіні. Компонент ігнорує більшість властивостей `StyleSheet`.
- **Жорстка структура:** Кнопка приймає лише текст через атрибут `title`. Ви не можете вставити всередину іконку або інший компонент.
- **Платформозалежність:** На **iOS** це просто кольоровий текст (без фону), а на **Android** — прямокутник із заливкою та тінню. Один і той самий код дасть візуально різний результат.
- **Фіксована висота:** Ви не можете напряму змінити розмір кнопки. Вона завжди підлаштовується під системні стандарти.



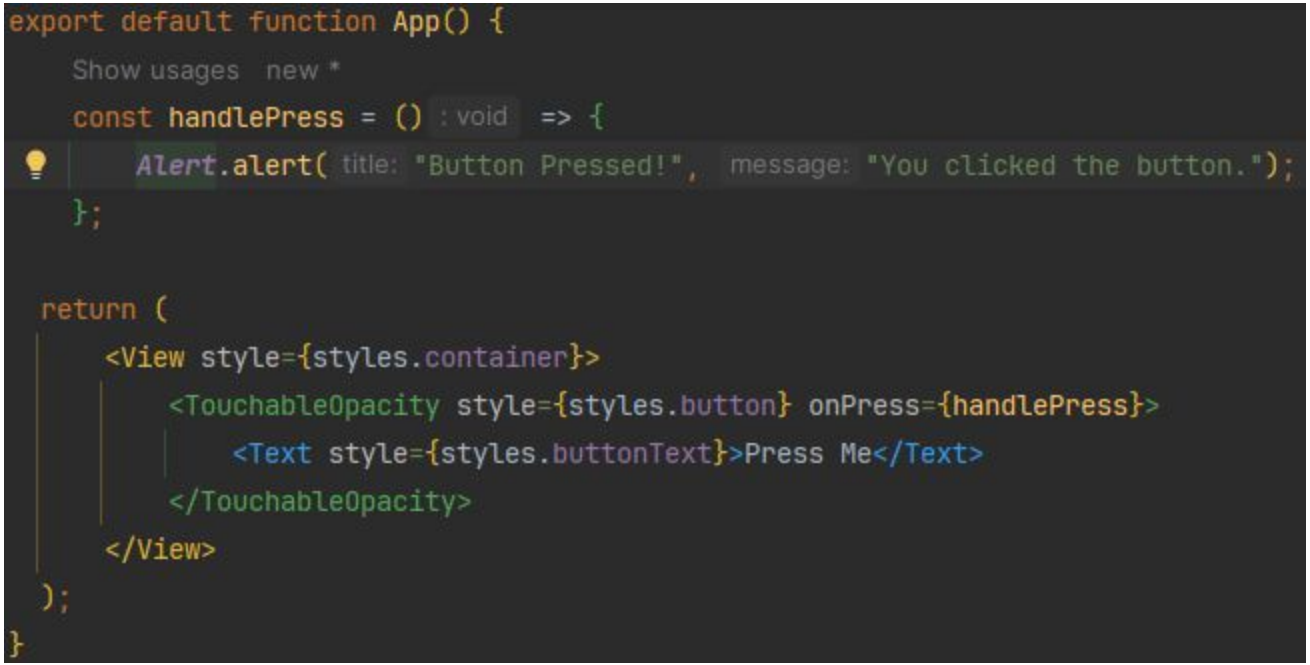
TouchableOpacity

Це компонент-контейнер, який реагує на натискання зміною прозорості (opacity)..На відміну від `Button`, тут немає жодних стилістичних обмежень:

- **Будь-яка форма**.
- **Будь-який вміст**.
- **Повна стилізація:** Підтримує всі властивості `StyleSheet` (flex, padding, borderRadius, shadows).

Ключові властивості

- **`onPress`**: Основна подія (аналог `onClick` у вебі).
- **`activeOpacity`**: Рівень прозорості при натисканні (від `0` до 1). За замовчуванням — `0.2`.
- **`onLongPress`**: Окрема подія для тривалого затискання кнопки.


export default function App() {
Show usages new *
const handlePress = () : void => {
Alert.alert(title: "Button Pressed!", message: "You clicked the button.");
};
return (
<View style={styles.container}>
<TouchableOpacity style={styles.button} onPress={handlePress}>
<Text style={styles.buttonText}>Press Me</Text>
</TouchableOpacity>
</View>
);
}

TouchableHighlight

`TouchableHighlight` — це інтерактивний контейнер, який при натисканні затемнює або змінює колір фону елемента. На відміну від `TouchableOpacity`, цей компонент працює шляхом додавання додаткового кольорового шару під ваш контент. Коли користувач натискає на нього, верхній шар стає прозорим.

Ключові властивості

- **`underlayColor`**: (Обов'язково) Колір, який з'явиться під час натискання. Зазвичай це трохи темніший відтінок основного фону.
- **`onPress`**: Функція обробки натискання.
- **`activeOpacity`**: Ступінь прозорості основного контенту під час натискання (зазвичай ставлять ближче до 1, щоб колір підкладки було добре видно).

`TouchableHighlight` вимагає, щоб всередині був **тільки один** дочірній елемент.

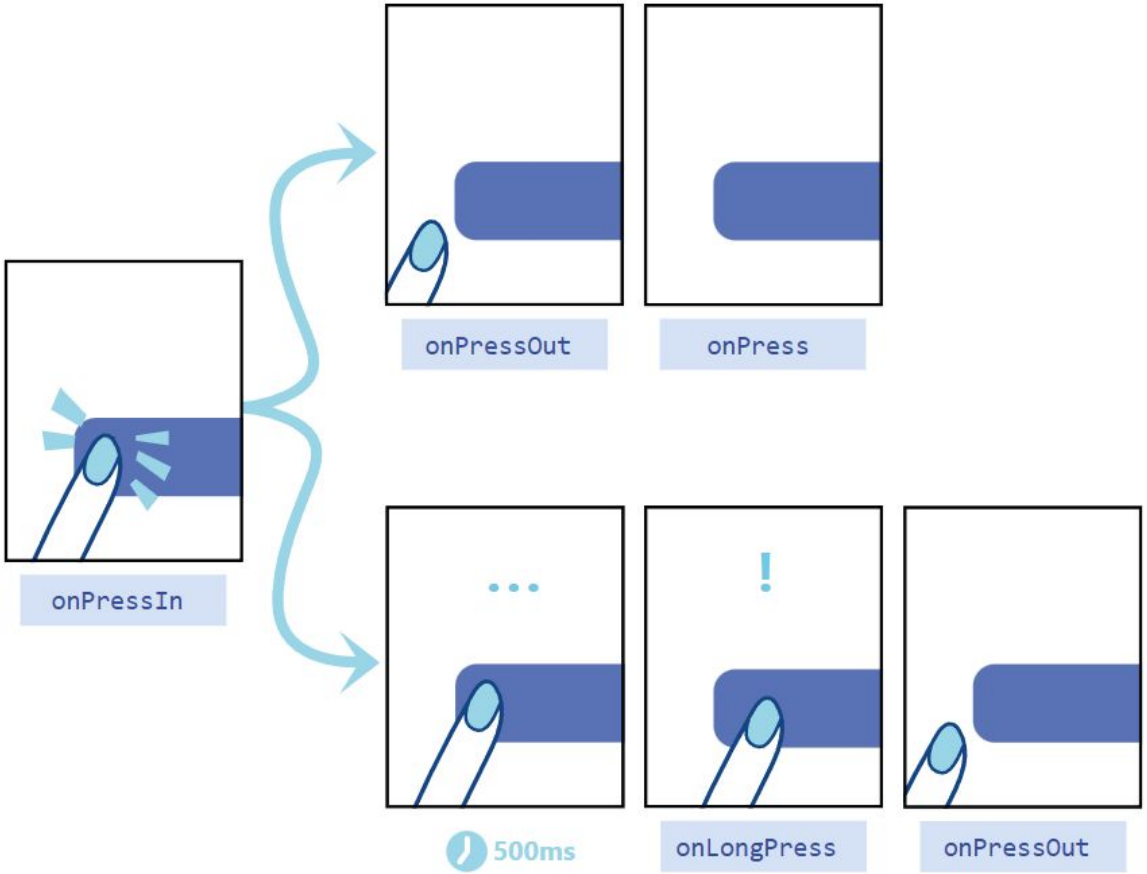
```
export default function App() {
    Show usages  new *
    const handlePress = () : void  => {
        Alert.alert( title: "Button Pressed!",  message: "You clicked the button.");
    };

  return (
      <View style={styles.container}>
          <TouchableHighlight
              style={styles.button}
              underlayColor="#2980b9"
              onPress={handlePress}
          >
              <Text style={styles.buttonText}>Press Me</Text>
          </TouchableHighlight>
      </View>
  );
}
```

Pressable у React Native

Pressable — це найновіший та найбільш гнучкий компонент-контейнер, що прийшов на зміну застарілим `Touchables`:

✔ Дає **повний контроль над стилями у різних станах натискання**.
✔ Підтримує **натискання (`onPress`), довге натискання (`onLongPress`) та відпускання (`onPressOut`)**.
✔ Дозволяє **змінювати стиль кнопки залежно від стану (`pressed`)**.
✔ Використовується для **створення кастомних кнопок, інтерактивних елементів, списків тощо**.

`Pressable` – це найгнучкіший компонент для взаємодії в React Native.


onPressOut
onPress
onPressIn
...
!
500ms
onLongPress
onPressOut

```jsx
<View style={styles.container}>
    <Pressable
        style={({ pressed : boolean }) => [
            styles.button,
            pressed && styles.buttonPressed // Додає стиль при натисканні
        ]}
        onPress={() : any => console.log("Button Pressed!")}
    >
        <Text style={styles.buttonText}>Press Me</Text>
    </Pressable>
</View>
```


```jsx
const styles : {...} = StyleSheet.create({
    container: {
        flex: 1,
        justifyContent: "center",
        alignItems: "center",
    },
    button: {
        backgroundColor: "#3498db",
        paddingVertical: 12,
        paddingHorizontal: 20,
        borderRadius: 8,
    },
    buttonPressed: {
        backgroundColor: "#b92957", // Колір при натисканні
    },
    buttonText: {
        color: "#fff",
        fontSize: 18,
        fontWeight: "bold",
    },
});
```

Робота з зображеннями в **React Native (Image)**

`<Image>` — це компонент для відображення статичних зображень, іконок та мережевих ресурсів. Це нативна обгортка, яка оптимізує рендеринг графіки на iOS та Android.

Джерела зображень (source)

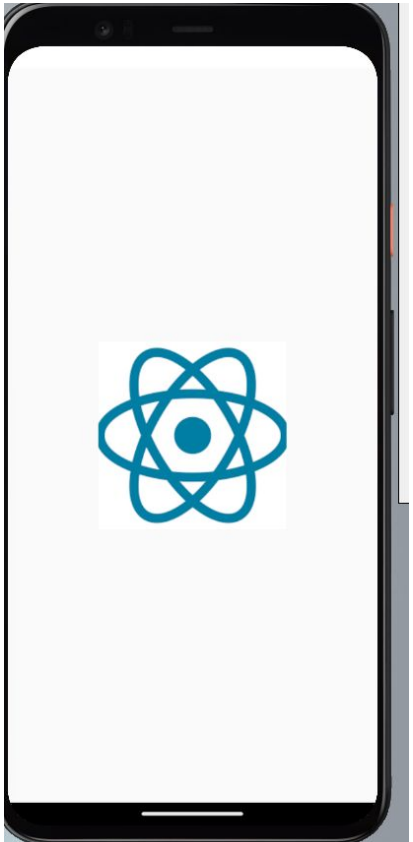
На відміну від HTML-тегу `<img>`, у React Native тип джерела визначає спосіб завантаження:

- **Локальні ресурси:** Використовується `require('./path/to/img.png')`. Розміри зображення визначаються автоматично під час збірки.
- **Мережеві ресурси:** Використовується об'єкт із посиланням: `source={{ uri: 'https://link.com/img.png' }}`.
  Важливо: Для мережевих зображень **обов'язково** потрібно вказувати `width` та `height` у стилях, інакше вони не з'являться на екрані.

Базовий приклад **Image** (локальне зображення)

Щоб завантажити локальне зображення, використовується `require()`.

```
  return (
    <View style={styles.container}>
      {/* Локальне зображення з папки assets */}
      <Image source={require('./src/assets/React.png')} style={styles.image} />
    </View>
  );
}

const styles : {container: {...}, image: {...}}  = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
  },
  image: {
    width: 200,
    height: 200,
  },
});
```



`resizeMode` – Як змінювати розмір зображення

Керування розміром: `resizeMode`

Це аналог `object-fit` у CSS. Він визначає, як зображення вписується в межі контейнера:

- **`cover`** (дефолт): Заповнює всю область, обрізаючи краї (зберігає пропорції).
- **`contain`**: Вміщує зображення повністю, залишаючи порожнє місце за потреби.
- **`stretch`**: Розтягує зображення рівно під розмір рамки (можливі спотворення).
- **`repeat`**: Повторює зображення мозаїкою (тільки для iOS).
- **`center`**: Розміщує картинку по центру без масштабування.

Обробка помилок (onError) та завантаження (onLoad)

onError дозволяє виводити альтернативний контент, якщо зображення не завантажилось. **onLoad** можна використовувати для **анімації при завантаженні**.

Приклад:

```
<Image
  source={{ uri: 'https://wrong-url.com/image.png' }}
  style={styles.image}
  onError={() => console.log('Помилка завантаження')}
/>
```

Використання **defaultSource** для кешування

`defaultSource` використовується для показу **зображення-заглушки**, поки завантажується основне.

Приклад:

```
<Image
  source={{ uri: 'https://reactnative.dev/img/tiny_logo.png' }}
  style={styles.image}
  defaultSource={require('./assets/placeholder.png')}
/>
```

Cache Control

Режими кешування:

- **default** – Використовує стандартну стратегію платформи.
- **reload** – Завантажує дані лише з джерела, ігноруючи кеш.
- **force-cache** – Використовує кешовані дані, навіть якщо вони застарілі. Якщо кешу немає, завантажує з джерела.
- **only-if-cached** – Використовує лише кешовані дані. Якщо кешу немає, завантаження не відбувається.

```
<Image
  source={{
    uri: 'https://reactjs.org/logo-og.png',
    cache: 'only-if-cached',
  }}
  style={{width: 400, height: 400}}
/>
```

Фонове зображення **ImageBackground**

У веб-розробці часто використовується `background-image`, але у React Native це реалізується через компонент **`ImageBackground`**.

- Має такі ж властивості, як і `Image`.
- Дозволяє вкладати інші компоненти поверх фону.

Приклад коду:

```
return (
  <ImageBackground source={...} style={{width: '100%', height: '100%'}}>
    <Text>Inside</Text>
  </ImageBackground>
);
```

```jsx
<ImageBackground
    source={{ uri: 'https://reactnative.dev/img/tiny_logo.png' }} // URL зображення
    style={styles.background}
    resizeMode="cover" // Заповнює весь контейнер
    imageStyle={{ opacity: 0.7 }} // Прозорість фону
>
    <Text style={styles.text}>Welcome!</Text>
    <Text style={styles.text}>Welcome!</Text>
    <Text style={styles.text}>Welcome!</Text>
    <Text style={styles.text}>Welcome!</Text>
    <Text style={styles.text}>Welcome!</Text>
</ImageBackground>
```

ScrollView

`ScrollView` — це універсальний контейнер для прокручування. Він дозволяє користувачу взаємодіяти з контентом, обсяг якого перевищує фізичні розміри екрана пристрою.

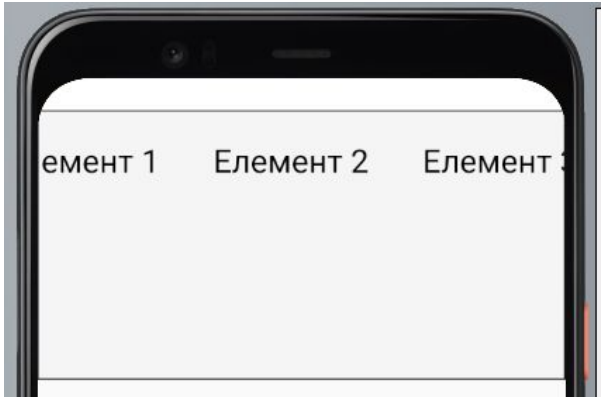
- Використовується для **невеликих списків** або контенту, що потребує прокручування.
- **Недолік**: `ScrollView` завантажує **всі елементи одразу**, що може спричинити проблеми з продуктивністю для довгих списків.

Базовий приклад використання **ScrollView**

```
<View style={styles.view}>
    <ScrollView style={styles.container}>
        <Text style={styles.text}>Елемент 1</Text>
        <Text style={styles.text}>Елемент 2</Text>
        <Text style={styles.text}>Елемент 3</Text>
        <Text style={styles.text}>Елемент 4</Text>
        <Text style={styles.text}>Елемент 5</Text>
    </ScrollView>
</View>
```

Горизонтальний **ScrollView**

```
<View style={styles.view}>
    <ScrollView
        style={styles.container}
        horizontal={true}
    >
        <Text style={styles.text}>Елемент 1</Text>
        <Text style={styles.text}>Елемент 2</Text>
        <Text style={styles.text}>Елемент 3</Text>
        <Text style={styles.text}>Елемент 4</Text>
        <Text style={styles.text}>Елемент 5</Text>
    </ScrollView>
</View>
```



Приховування смуги прокрутки

Щоб **сховати вертикальний або горизонтальний індикатор прокрутки**:

- **`showsVerticalScrollIndicator={false}`** – прибирає **вертикальний скролбар**.
- **`showsHorizontalScrollIndicator={false}`** – прибирає **горизонтальний скролбар**.

Прокрутка до певного місця (**scrollTo**)

Іноді потрібно **програмно прокручувати** `ScrollView` до певного місця. Це можна зробити за допомогою **`scrollTo`**.

```tsx
export default function App() {
    const scrollViewRef : MutableRefObject<null> = useRef( initialValue: null);

    Show usages  new *
    const scrollToBottom = () : void => {
        scrollViewRef.current.scrollToEnd({ animated: true }); // Прокрутка вниз
    };

    return (
        <>
            <ScrollView ref={scrollViewRef} style={styles.container}>
                {[...Array( arrayLength: 20)].map((_, i : number ) => (
                    <Text key={i} style={styles.text}>Елемент {i + 1}</Text>
                ))}
            </ScrollView>
            <Button title="Вниз" onPress={scrollToBottom} />
        </>
    );
}

const styles : {container: {...}, text: {...}} = StyleSheet.create({
    container: { flex: 1, backgroundColor: '#f5f5f5' },
    text: { fontSize: 24, margin: 20 },
});
```

RefreshControl — Оновлення через свайп (Pull-to-Refresh)

`RefreshControl` — це компонент, який дозволяє користувачу оновлювати вміст екрана, потягнувши його вниз (жест Pull-to-Refresh). Він інтегрується всередину компонентів, що мають скрол (`ScrollView` або `FlatList`).

Як це працює?

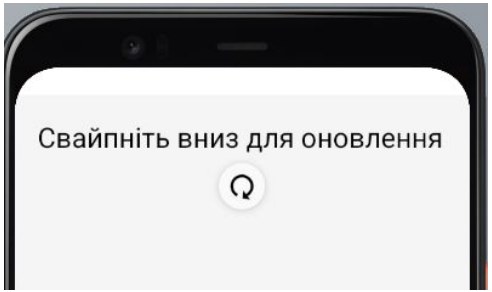
Коли користувач тягне контент вниз, з'являється системний індикатор завантаження. Після завершення жесту викликається функція оновлення даних, а індикатор зникає, коли нові дані отримано.

Ключові властивості (Props):

- **`refreshing`**: Булеве значення (state). Визначає, чи відображається індикатор завантаження в цей момент.
- **`onRefresh`**: Функція, яка запускається при виконанні жесту. Зазвичай вона ініціює запит до API або оновлює масив даних.
- **`colors` (Android)**: Масив кольорів для анімації індикатора.
- **`tintColor` (iOS)**: Колір індикатора (спінера) для пристроїв Apple.

Використання **refreshControl** (оновлення при свайпі вниз)

```
Show usages  ▲ kipz_nsi *
export default function App() {
    const [refreshing : boolean , setRefreshing] = useState( initialState: false); // Стан оновлення

    Show usages  new *
    const onRefresh = () : void  => {
        setRefreshing( value: true); // Починаємо оновлення
        setTimeout( handler: () : void  => setRefreshing( value: false),  timeout: 2000); // Емуляція оновлення (2 сек)
    };

    return (
        <ScrollView
            refreshControl={
                <RefreshControl refreshing={refreshing} onRefresh={onRefresh} />
            }
            style={styles.container}
        >
            <Text style={styles.text}>Свайпніть вниз для оновлення</Text>
        </ScrollView>
    );
}
```

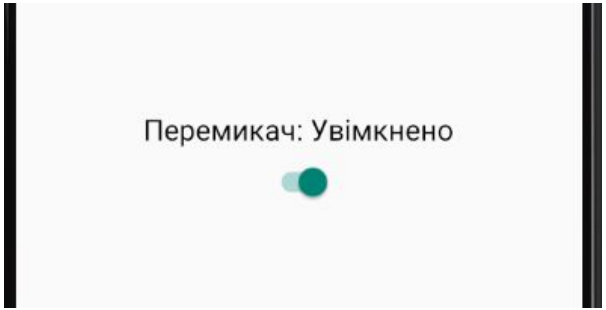


Switch

`Switch` – це компонент у **React Native**, який використовується для **перемикання між двома станами (увімкнено/вимкнено)**.

- **Аналог чекбокса або тумблера у веб-розробці.**
- Використовується для **налаштувань**, **темної/світлої теми**, **увімкнення функцій** тощо.

```
export default function App() {
    const [isEnabled : boolean , setIsEnabled] = useState( initialState: false); // Стан перемикача

    Show usages  new *
    const toggleSwitch = () : void  => setIsEnabled( value: previousState : boolean  => !previousState)

    return (
        <View style={styles.container}>
            <Text style={styles.text}>
                Перемикач: {isEnabled ? 'Увімкнено' : 'Вимкнено'}
            </Text>
            <Switch value={isEnabled} onValueChange={toggleSwitch} />
        </View>
    );
}
```



Кастомізація кольорів

Можна змінювати колір перемикача в різних станах.

```
<Switch
  value={isEnabled}
  onValueChange={toggleSwitch}
  thumbColor={isEnabled ? 'white' : 'gray'} // Колір кнопки
  trackColor={{ false: 'red', true: 'green' }} // Колір фону
/>
```

Результат:

- Коли `Switch` **увімкнено**, колір треку – зелений.
- Коли `Switch` **вимкнено**, колір треку – червоний.
- Кнопка змінює колір залежно від стану.

Modal

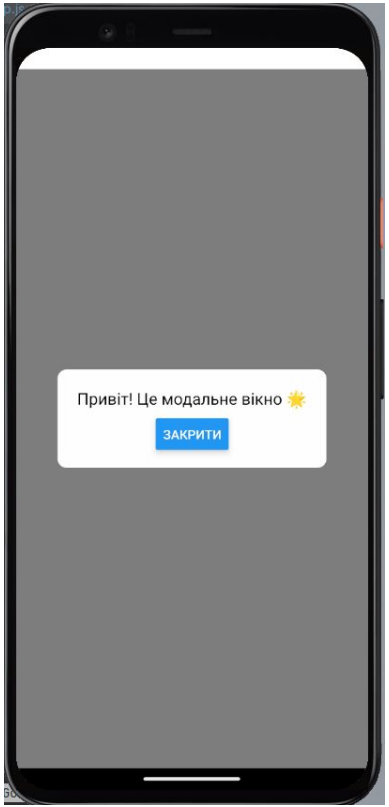
`Modal` — це вбудований компонент для відображення контенту поверх основного інтерфейсу. Це ідеальний інструмент для сповіщень, форм вводу або вибору налаштувань.

Основні принципи:

- **Контрольованість:** Відображення модального вікна повністю залежить від пропса `visible` (булеве значення).
- **Платформозалежність:** На iOS та Android модалки мають різні системні анімації та поведінку.
- **Перекриття:** Modal завжди рендериться поверх усіх інших компонентів у дереві (навіть якщо він прописаний внизу коду).

Базовий приклад використання **Modal**

```tsx
export default function App() {
    const [modalVisible : boolean , setModalVisible] = useState( initialState: false); // Стан модального вікна

    return (
        <View style={styles.container}>
            {/* Кнопка для відкриття модального вікна */}
            <Button title="Відкрити модальне вікно" onPress={() : void => setModalVisible( value: true)} />

            {/* Модальне вікно */}
            <Modal visible={modalVisible} animationType="slide" transparent={true}>
                <View style={styles.modalContainer}>
                    <View style={styles.modalContent}>
                        <Text style={styles.text}>Привіт! Це модальне вікно 🌟</Text>
                        <Button title="Закрити" onPress={() : void => setModalVisible( value: false)} />
                    </View>
                </View>
            </Modal>
        </View>
    );
}
```



`animationType`: Анімація відкриття модального вікна

Властивість `animationType` визначає візуальний ефект під час відкриття та закриття модального вікна. Це дозволяє зробити інтерфейс більш "живим" та нативним для користувача.

Доступні значення:

- **`none`**: Вікно з’являється миттєво. Використовується для технічних екранів, де швидкість важливіша за естетику.
- **`slide`**: Вікно плавно виїжджає знизу вгору.
  - *Best Practice:* Стандарт для **iOS**, нагадує системні картки налаштувань.
- **`fade`**: Вікно з’являється через плавне збільшення прозорості (ефект розмиття/проявлення).
  - *Best Practice:* Рекомендовано для **Android**, виглядає як класичне системне сповіщення.

`transparent`: Прозорий фон для модального вікна

Властивість **`transparent`** визначає, чи буде модальне вікно повністю перекривати попередній екран непрозорим фоном, чи дозволить бачити контент під ним.

Значення:

- **`false` (за замовчуванням):** Модальне вікно має білий (або системний) непрозорий фон. Попередній екран повністю зникає з поля зору.
- **`true`:** Фон модального вікна стає прозорим. Це дозволяє створювати напівпрозорі оверлеї, через які видно основний інтерфейс додатка.
-

```
<Modal visible={modalVisible} transparent={true}>
  <View style={{ backgroundColor: 'rgba(0,0,0,0.5)', flex: 1 }}></View>
</Modal>
```

ActivityIndicator

`ActivityIndicator` — це стандартний системний компонент, який показує користувачеві, що в додатку триває якийсь процес (завантаження даних, обробка платежу тощо).

- Використовується для **індикації завантаження даних, очікування запитів API** тощо.
- Доступний **за замовчуванням у React Native** (не потрібно встановлювати додаткові бібліотеки).
- Працює **як на iOS, так і на Android**.

Базове використання **ActivityIndicator**

```tsx
export default function App() {
    const [loading : boolean , setLoading] = useState( initialState: false); // Стан завантаження

    return (
        <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
            {/* Якщо loading === true, то показуємо ActivityIndicator */}
            {loading && <ActivityIndicator size="large" color="green" />}

            {/* Кнопка, яка вмикає індикатор */}
            <Button title="Завантажити" onPress={() : void => setLoading( value: true)} />
        </View>
    );
}
```